An Analysis of Composability and Composition Anomalies

Lodewijk Bergmans, Bedir Tekinerdogan,
István Nagy & Mehmet Aksit

TRESE Software Engineering group,
Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{bergmans, bedir, nagyist, aksit}@cs.utwente.nl

Abstract

The separation of concerns principle aims at decomposing a given design problem into concerns that are mapped to multiple independent software modules. The application of this principle eases the composition of the concerns and as such supports composability. Unfortunately, a clean separation (and composition of concerns) at the design level does not always imply the composability of the concerns at the implementation level. The composability might be reduced due to limitations of the implementation abstractions and composition mechanisms. The paper introduces the notion of *composition anomaly* to describe a general set of unexpected composition problems that arise when mapping design concerns to implementation concerns. To distinguish composition anomalies from other composition problems the requirements for composability at the design level is provided. The ideas are illustrated for a distributed newsgroup system.

Keywords: Composition, composability, composition anomaly, inheritance anomaly, analysis.

1. Motivation and Overview

It is generally acknowledged that the "separation of concerns" is one of the crucial issues in achieving adaptable, reusable and maintainable software [14] . This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability. The principle can be applied in various ways and one can safely state that the separation of concerns principle is a ubiquitous software engineering principle.

The term *concern* generally refers to concepts within an application that are relevant to that application (i.e. that the application is 'concerned with'). Each concern has usually incarnations throughout the life cycle of a system; i.e. there are requirements relating to it, analysis and design abstractions and implementation level representations as well (e.g. classes, aspects [15], hyperslices [27], modules, packages, methods, code fragments).

Identifying and separating concerns is a difficult design activity in itself, which requires extensive knowledge of the domain. Furthermore, even when all concerns have been cleanly separated and specified, one must still define how the individual concerns are to be composed to form a system that meets its requirements.

The transition from design to implementation[1] involves mapping the design-level concerns and compositions to implementation level concerns and compositions. Implementation languages support only a restricted set of composition mechanisms (e.g. inheritance, aggregation, aspect weaving, etc.), which have very concrete and often unique semantics. Concerns at the design level will be mapped to –parts of– one or more concerns at the implementation level. It turns out that composability at the design level does not directly result in composability at the implementation level. Given the neat separation of concerns and the composability, albeit expected, composability might be impeded at the implementation level due to the limitations of the composition operators. We term these kinds of problems as *composition anomalies.*

One of the goals of this paper is to offer improved insights into composability. This may benefit all software engineers that encounter composition problems in practice. In particular, this paper explains when composition problems are due to an inadequate design and when they are due to the properties of the adopted composition model at the implementation level (typically defined by the programming language). This knowledge can directly support the design of more composable systems. We also think that a better understanding of composability can benefit the design of models and mechanism for the composition of software.

This paper should help software engineers to gain a better understanding of the composition problems that they encounter. It can help researchers to understand software composition better and especially to help them to design composition models and languages that do not suffer from composition anomalies.

The structure of this paper is as follows: Section 2 presents the terminology that we adopt in discussing composability. Section 3 provides a conceptual model on composition which will be used to explain the composability requirements at the design level in section 4. In section 5 we introduce the so-called *composition anomalies*. Briefly, these are composition problems that can eventually be attributed to limitations of the implementation model. To be able to reason about this, it is important to consider properties of the input first: to distinguish between composition failures that are due to design flaws and those that are due to the applied composition scheme, it is necessary to formulate a set of criteria for judging design level composability. If these criteria are not fulfilled, a composition will not succeed, regardless of the composition scheme. We conclude this paper by putting it in the context of related work, in section 6, and summarizing the contribution of this paper in section 7.

2. Background and Terminology

The composition of software has been studied extensively throughout the history of computer science. Well-known early publications are for example [14][25] who discuss the application of separation of concerns in order to support composability. Accordingly, many composition techniques have been proposed; consider for example various programming paradigms and their associated composition techniques such as procedural programming (procedure calls), functional programming (function composition), object-oriented programming (inheritance, object aggregation and message invocation) and a whole range of proposals for module mechanisms within each of these paradigms (e.g. as in Modula-2 and Ada . More recently, aspect-oriented software development (AOSD) has been proposed to cope with the so-called crosscutting concerns. AOSD introduces a new composition

[1] This is typically the main goal of 'detailed design'. Note that this discussion of the relation between the design level and the implementation level can be repeated for other levels as well.

technique called aspect weaving [15]. From this discussion it becomes clear that composition techniques play a major role in programming paradigms.

In explaining our ideas we will adopt the following definitions:

---

definition: *composition*

1. The act of applying one or more *composition operators* on a given set of concerns, or

2. The result of a composition process (as in "C_r is the composition of C_1 and C_2").

This definition implies that composition is either an act or a result of the act. When composing concerns a composition operator is applied (see next definition)

---

definition: *composition operator*

A *composition operator* is a function that takes concerns as input, and produces a new concern (explicitly or implicitly visible) that combines the input concerns in whole or part.

Clearly, this is a very broad definition, which covers a wide range of composition techniques. Typically, this means that the new, composed, concern exhibits behavior and/or structure from all the input concerns. An important category is where the composed concern exhibits *all* the observable behavior of the individual input concerns (unless conflicting or explicitly defined otherwise). Inheritance is a typical example: single inheritance takes two concerns (a class C_1 and an ancestor class C_2), and composes these into a new class (not explicitly visible), which we will call C_1'. Instances of this class combine behavior and properties from both C_1 and C_2.

Both the input and the output of the compositions consist of concerns. In order to reason about concerns we introduce the notion of concern model:

---

definition: *concern model*

A *concern model* represents the common structure of the concerns that are used in the composition;

Note that every programming paradigm adopts a different concern model. For example, the structured programming paradigm includes the concern model that represents concerns as operations, the object-oriented programming paradigm represents concerns as objects, the aspect-oriented programming paradigm represent concerns (in addition) as aspects, etc.

Similar to a general representation for composition we can also model compositions which we define as composition scheme:

---

definition: *composition scheme*

A composition scheme is a model of composition, representing a set of composition operators that must all be applicable for a specific concern model.

As such, each programming paradigm has a different composition scheme because of the different concern model, but also since different composition operators are used. The object-oriented paradigm applies, for instance, aggregation and inheritance as composition operations; the aspect-oriented paradigm enhances this with the composition operator for weaving.

Programming paradigms do not only reflect a way of thinking but also improve the solutions to problems encountered in the earlier programming paradigms. One problem, for example, in the object-oriented paradigm is the possible occurrence of the so-called inheritance

anomalies that can appear in case of concerns, such as synchronization. Matsuoka et al [20][21] described and analyzed cases where adding synchronization code to classes caused serious maintenance problems when trying to reuse and extend such code through inheritance mechanisms. In those cases, the problems typically appeared as a need to frequently override methods by copying substantial parts of the methods of the superclasses. In our own work we have also shown that the aggregation operator requires several redefinitions in case of crosscutting concerns [4][1][2].

The term anomaly means a "deviation from the common rule; irregularity" or "something different, abnormal, peculiar, or not easily classified" [31].

In this paper we introduce the notion of *composition anomalies*. The term, as informally stated, refers to the cases where a composition is intuitively and conceptually sound at the design level cannot be realized correctly by a specific composition scheme at the implementation level.

3. A Conceptual Software Composition Model

Complex systems often include interrelated subsystems which are again a composition of hierarchically structured (sub-) subsystems, until the lowest, atomic level [26]. At each level, the subsystems of a system tend to be interrelated and mutually dependent; in fact, no interesting complex systems can be built from fully independent, isolated subsystems. Therefore, to build complex systems it is important that we understand composition and composability.

In reasoning about composition we distinguish between two different composition views: *operational view* and *specification view*:

- The *operational view* of a system is close to a physical view of the real world, and of the executing software as well. For example, physical and biological systems describe the structure of systems from an *operational view*. Typically, the composition from an operational view structures represents a snapshot of the operating system.

- In the *specification view* the structures are more stable and represent basically the static structure. Software engineers have to extensively specify compositions before they are built and as such have to adopt a *specification view.*

In composing complex system very often both views are applied. In UML, for example, most of the system modeling is done using class diagrams, defining the *specification view.* On the other hand, object and collaboration diagrams show an *operational view.*

In addition both views can be applied throughout the software development life cycle. For the sake of the discussions in this paper, we distinguish (just) two levels of abstraction of a system: the *design level* and the *implementation level*. We assume that the design level abstracts from the details of programming and programming languages. The implementation level adopts a specific programming language and/or programming technology, and is required to add all details to come to executable specifications, such as writing the bodies of methods.

Based on the discussion above Figure 1 presents a conceptual model of the essential properties for discussing software composition. The model defines the composition as a space consisting of four spaces. A composition is represented as a graph in which the nodes represent the concerns and the edges the composition relations. The nodes can be either *concern types* or *concern instances*. Concern types stand for the concerns in the specification

view and are represented as rectangles. Concerns instances are the concerns in the operational view and are represented through ovals.

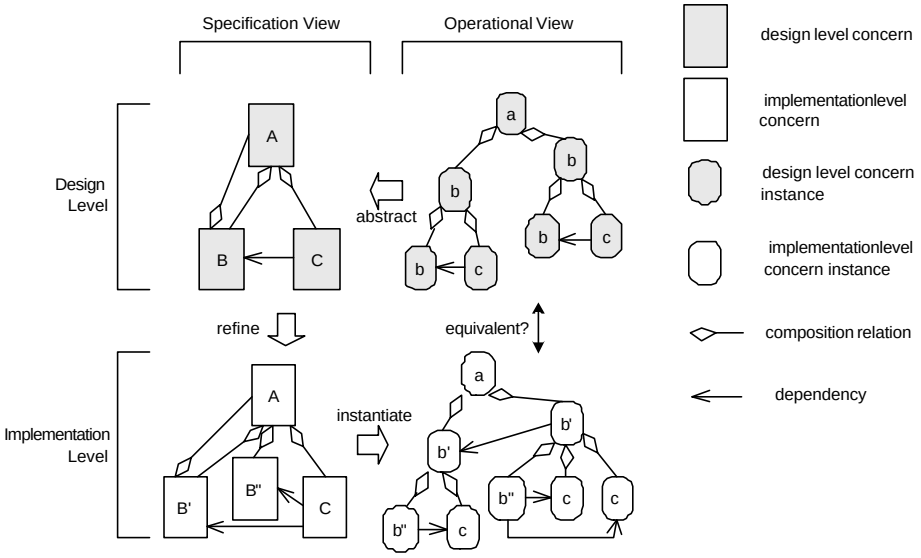


Figure 1. Four views upon software composition.

Every composition could be in principle represented as defined in this model. We will now briefly discuss each of the four views in Figure 1:

- *Design level Operational View*: this can be seen as a representation of the real world. It shows concern instances, which are composed hierarchically. Concern instances may have further dependencies (such as interactions or references) with other instances.
- *Design Level Specification View*: this is a design level model of a system; UML class diagrams at an analysis or early design phase are a typical example.
- *Implementation Level Specification View*: a model of the system in terms of the programming language or execution system. The model consists of concern specifications that will be instantiated and composed during execution. This also involves choosing a particular form to represent the design level concern, such as module, class, instance variable, method or even code fragment (from a method body).
- *Implementation Level Operational View*: The total (sub-)system, designated by the root concern instance, is constructed as a nested (hierarchical) composition of concern instances. The concern instances may have one or more dependencies with (arbitrary) other concern instances, although ideally these dependencies are very local; only with and between children.

The four views are not independent from each other but are related to each other through the following operations:

- *Abstracting* from the operational view to the specification view involves mapping the concern instances to concerns that define their properties.
- *Refining* involves the transformation of the design level specification such that it can be expressed in terms of the implementation language. Examples of typical issues are the conversion of associations in UML, converting multiple inheritance into single inheritance, etc.

- *Instantiating* the implementation level specification means that the concerns of the specification will be used as templates to create multiple instances with specific property values.

Besides of these dependencies there is also a dependency between the *design level operational view* and the *implementation level operational view.* We claim that, for optimal composability of the concerns that make up a system, the structure of the implementation level operational view should be isomorphic to the structure of the design level operational view. This means that each concern at the design level can be mapped to a corresponding entity at the implementation level, thereby preserving the structure of the concerns at the design level. This is based on the separation of concerns principle in which all relevant concerns in the design need to be represented as first-class entities in the implementation. The advantage of this is that they can evolve and be adapted independently.

We will elaborate on this composition model in the subsequent sections and discuss the requirements for composability and give an analysis of composition anomalies.

4. Composability Requirements

Composability is a quality characteristic that designates 'the ability for composition'. The composability of a given set of concerns is determined by both the concerns and the *composition scheme* that is used for the composition. We defined a *composition scheme* as a set of distinct composition operators. Composability is defined as the ease of composing a set of concerns into a new concern. The degree of composability depends both on the properties of the concerns and on the suitability of the available composition operators.

Based on the conceptual model for composability it is possible to discuss composability both at the design level and implementation level. To ensure implementation composability it is required that design composability is ensured. Therefore, in this section we define rules for *design composability*. We assume that a given composition that adheres to these rules will not suffer from design problems. As such we can more precisely pinpoint the problems that are caused to composition anomalies as described in the following section.

To explain the rules we will use the following example:

Example:

A Distributed Newsgroup System (DNS) allows people to communicate and share their experiences via Internet through the use of electronic newsgroups. Users may create a newsgroup or subscribe to existing newsgroups. To this aim, the system classifies the newsgroups into different categories. Each group has a profile, which is determined by the owner/creator of the group. Each group has a specific profile indicating its description and access properties. In addition to the owner and the regular members, groups might have moderators, which are assigned by the owner. The group owner also determines the access rights of the moderators. The administrator determines all the system limitations and capabilities including the rights of owner/moderator/member, and group profile options. Within a newsgroup, members can mail, upload files, send photos, define tables in a (simple) database, use a calendar, etc. The system allows its users to set their user and member profiles. User profiles simply include the personal information about the user, such as the first, last name, address, e-mail address, etc. On the other hand, a member profile represents the choices of a member for one group.

Figure 2 represents a part of the newsgroup architecture. The component *NewsgroupActor* represents the actors in the newsgroup system. These are *Administrator*, *Owner*, *Moderator* and *Member*. Each Newsgroup Actor owns a *Profile* that is defined through a *Profile Manager*.

Actors can interact with the newsgroup system by sending requests to *TransactionManager*, which will utilize *SecurityManager* to check for authentication and authorization; if the request is accepted it will be forwarded to *DataManager*. *DataManager* is responsible for the consistency of the newsgroup objects. To do this, it applies concurrency control and recovery techniques.

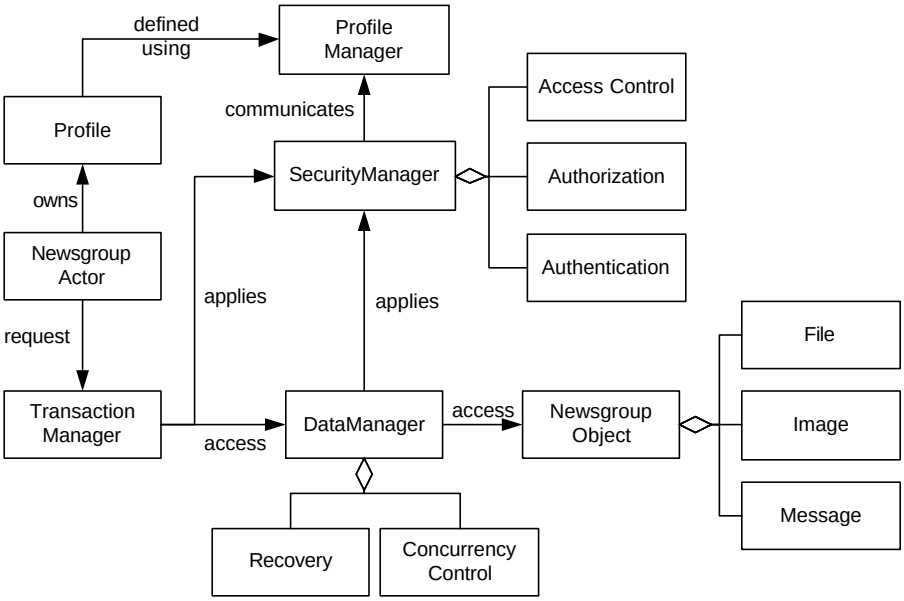


Figure 2. Part of the Newsgroup System Architecture

Every composition is required to provide some function of value. This may be needed, for example, to enhance existing functionality or to define more complex behavior:

1. *Composition must be conceptually sound.*

   The composition of a given set of concerns must be conceptually sound, or 'relevant'. This means that the overall composition must fulfill or support a predefined goal. Typically this goal can be derived from the requirement specification and the solution domain. The requirement specification defines what is desired, while the solution domain defines what is possible. A composition is only sound if it is valid with respect to both the requirement specification and the solution domain. This relates to composability at the conceptual level.

   In the DNS example a given composition is sound whenever it is explicitly described in the requirement specification or can be identified in the solution domain of distributed newsgroup systems. For example, composing the data management with a concurrency control mechanism can be considered as a sound composition because a DNS is by nature a concurrent system and all data access must be synchronized.

2. *Components must be syntactically compatible*

   We assume that each component is defined as a set of operations that can be called by other components. In each operation another operation can be called. In order to be able to communicate with each other two components must firstly define the same syntactical operations. If two components are syntactically compatible we define this as *syntactic composability*.

In the given example syntactic composability typically means that the interfaces of the newsgroup components must be known to the actors and be aligned with the components of the actors. Further, the components of the DNS will also interact with each other as well. For example, the component *DataManager*, which is responsible for controlling access to the newsgroup objects can be composed with *SecurityManager*, which is responsible for authentication and authorization issues. To check whether a user has the access rights to a given file *DataManager* and *SecurityManager* need to communicate. This requires that the interface of the *DataManager* is syntactically compatible with the interface of *SecurityManager*.

3. *Components must be semantically compatible*

Semantic compatibility is needed to ensure that all components include the right semantic behavior. Note that this is orthogonal to syntactic compatibility. Components could be semantically compatible but lack the right interfaces. On the other hand, syntactically compatible components might not be composable because they have semantics that may result in conflict when they are composed.

In the example *DataManager* includes the components *Recovery* and *Concurrency Control* for ensuring consistent access to the newsgroup objects. Both components are conceptually sound and in addition, we could ensure that the interfaces are syntactically correct. The composition of these components will yield though a new concept and as such must be checked again for soundness. It has been shown in **Error! Reference source not found.**, for example, that some recovery techniques conflict with particular concurrency control techniques. When composing the recovery and concurrency control components this semantic conflict must be taken into account to ensure a semantically correct composition.

4. *Components must adhere to the same protocol*

A composition is not only static but also includes dynamic behavior. To function appropriately it is required that the corresponding components adopt the same communication protocol. The communication protocol basically refers to the partial ordering of the operations and the blocking conditions. This can also be described as the semantics of the interaction.

5. *Composition of components must be synchronized in time*

Compositions can take place at different times; edit time, compile time, debugging time, run-time, etc. This is in the first place determined by the composition operator. For example, inheritance is normally a compile-time operator. In order to succeed with the composition all the necessary components must have the same *time-compatibility*. For example, creating a new class (as inheritance does), does normally only make sense at compile time, since classes need to be compiled before execution.

In the given example, we could already have a component *DataManager* but the component *SecurityManager* might not be available yet. Even though both components might be syntactically and semantically compatible, for the eventual composition it is required that both components are available at the same time.

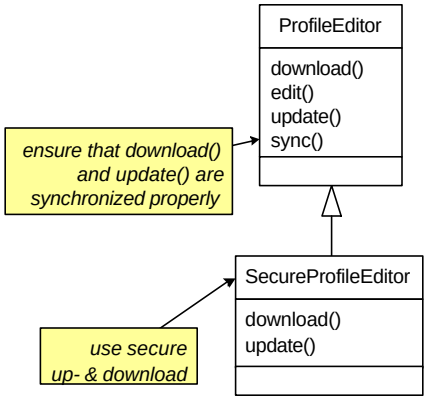
5. Composition Anomalies

In this section we will explore the notion of composition anomaly in more detail. In section 5.1 we provide a definition of composition anomaly, and illustrate this with an example. In section 5.3 we discuss the essence of composition anomaly, based on the model we introduced in section 3.

5.1 An Introduction to Composition Anomaly

Intuitively, a composition anomaly is a situation where a composition that is, or seems, perfectly right at the design level, does not yield the desired result at the implementation level.

To illustrate the meaning of composition anomaly, consider the following simple example of composition through inheritance in our DNS example: users can upload and download profiles. If a user downloads a profile for editing, no other user is allowed to do the same. This requirement can be realized by composing synchronization with the profile editing functionality. If an object-oriented design model is used, the solution will typically be as represented in figure 3; here, the design level composition is expressed using UML, the implementation level as Java code:




```
class ProfileEditor extends EditorTool{
   void download(Name profileName){
      this.sync(); // first check synchr.
      …  // perform actual download
   }
   … // other methods
}
class SecureProfileEditor extends ProfileEditor{
   void download(Name profileName){
      // perform actual download & decrypt
      … decrypt(profile, theKey) … ;
   }
   … // other overridden methods
```

Figure 3. Example of composition anomaly: (a) design level, (b) implementation level

In this example, at the design level a *sync()* method is declared that defines synchronization constraints between multiple *download()* and *update()* requests. When defining a subclass that redefines the *download()* method, this is seemingly fine. However, because of the way in which at the implementation level the synchronization (i.e. *sync()* method) is composed with the *download()* and *update()*, simple overriding of the these methods without taking synchronization into consideration, creates a faulty system.

Naturally, a software engineer may observe this issue and address it by adding another call to *sync()* in the appropriate methods of the *SecureProfileEditor*. Although this ensures that the system functions correctly, it introduces (or emphasizes) a maintainability problem: a lot of places in the code become dependent on the *sync()* method. This means that changes to the interface of the *sync()* method will have a large impact on the system. For example, if the original *sync()*should be replaced by a *sync_before()* and a *sync_after()* method, which must be called both (respectively at the start and at the end of a method), this requires rewrites of all the involved methods. Without going into details, we would like to point out that this problem, caused by too many dependencies, can be avoided through different composition operators. In particular, aspect weaving is an approach that allows for handling the same design in such a way that the dependencies are reversed and localized.

Similarly, one may consider the situation where synchronization at the *ProfileEditor* class is implemented with the Java keyword *synchronized*, causing methods to execute mutual exclusively (as expected). However, in the case one wants to loosen this synchronization constraint in a subclass, reimplementing *sync()* will not work as might be expected from the design.

Clearly, in these examples, the problems with the composition are caused by (subtle) differences between the composition at the design level and at the implementation level. In

these examples, these problems are rather easy to observe, but this may no longer be the case as the size of a system grows and/or the implementation details of inherited classes are not available. Also note that small discrepancies between the models at the design level and the implementation level are the norm rather than an exception.

5.2 Definition of Composition Anomaly

We will now define the term *composition anomaly* more precisely as follows:

---

definition: *composition anomaly*

Assume that:

- Given a design level composition $\oplus^d(c_1, c_2, .. c_n)$ that is *design composable*,
- and a given implementation level composition scheme $\odot$, which is suitable for the concern model of $c_1, c_2, .. c_n$
- and an implementation level composition operator $\oplus^i$ from the composition scheme $\odot$ that *corresponds* to $\oplus^d$.

The term *composition anomaly* is used to indicate those cases where –typically unexpectedly[2]– $\oplus^i(c_1, c_2, .. c_n)$ does not yield the *desired equivalent* of $\oplus^d(c_1, c_2, .. c_n)$.

We will further detail the meaning of the terms '*corresponds*' and '*desired equivalent*' that we used in this definition:

---

definition: *correspondence* between $\oplus^d$ and $\oplus^i$

- The *correspondence* between the design level composition operator $\oplus^d$ and the implementation level composition operator $\oplus^i$ means that the same intuitive and informal meaning is given to both; e.g. they both define inheritance, aggregation, or aspect weaving.

In fact, subtle differences in the (level of detail of the) semantics between these two levels are an important reason for composition anomalies. One of the typical effects of such differences is that the implementation level model is enhanced with one or more specifications to bridge those differences.

---

definition: *(no) desired equivalent*

Two composition operations $\oplus^d$ and $\oplus^i$ do *not* yield the *desired equivalent*, if:

(a) The composition process of $\oplus^i$ fails and yields no result, or

(b) the result of the implementation composition $\oplus^i$ does not exhibit the same observable behavior as the design level model: *behavior*($\oplus^i(c_1, c_2, .. c_n)$) ? *behavior*($\oplus^d(c_1, c_2, .. c_n)$), or

(c) the result of the implementation composition $\oplus^i$ does not have the same dependencies as the design level model: *dependencies*($\oplus^i(c_1, c_2, .. c_n)$) ? *dependencies*($\oplus^d(c_1, c_2, .. c_n)$), or

(d) the implementation composition $\oplus^i$ introduces code replication.

---

[2] The word *anomaly* typically applies to 'abnormal' or unexpected situations; this is obviously a subjective notion, hence we have broadened the definition of anomaly to all cases where the chosen implementation composition scheme cannot express the compositions that were expressed at the design level, ignoring the issues whether this could be expected (with certain pre-knowledge), or not.

The above cases (a) and (b) both introduce an implementation that –without taking measures– does not satisfy the functional requirements of the system. In many situations, these case can be (are) fixed through additional ('work-around') code. However, that usually leads to the cases (c) and (d), where almost always, code replication implies the creation of replicated dependencies. These cases do not directly affect the functionality of the system, but its maintainability; future changes are hard to make, or may have undesirable effects on the behavior.

For each of these cases, some examples can be given; (a) is illustrated by attempts to express multiple inheritance in a language like Java or Smalltalk, which do not support this; (b) was illustrated in the previous section when the calls to *sync()* where omitted in the subclass *SecureProfileEditor*; (c) and (d) both occur when adding calls to *sync()* in all the necessary places; this –simples possible– code fragment is replicated (d), adding many additional dependencies to the system (c).

5.3 The Essence of Composition Anomaly

In this section we will explain when certain compositions that are perfectly sensible at the design level, cause a composition anomaly at the implementation level, and why this depends on the specific composition operator that was adopted. We will illustrate this using the example from section 5.1 about the *ProfileEditor*, and showing how this example would appear in the various perpectives of the conceptual model that we introduced in section 3.

The resulting picture is shown in Figure 4. The design level specification view is similar to the UML diagram shown before, with the addition of the depenency relations between the *sync()* method and the methods that synchronization should be applied to. At this level, the *sync()* method depends on several other methods, since it needs to refer to each of them to apply the synchronization constraints. The operational view is similar, but makes all the compositions explicit; inheritance is here modeled as a relation between two instances, all the methods and attributes are modeled as first-class entities themselves, coupled to the instances through aggregation. Overridden methods are shown in gray text. Finally, the dependencies between the methods are drawn, in this case specifically for the case of inheritance; so the dependencies point to the methods in the subclass, if applicable.

The problems associated with composition anomaly are formed when moving from the design level composition operators to the implementation level operators: These may have slightly different semantics, or include semantic properties that are simply not considered at the design level. This is illustrated in the example by the fact that –e.g. in a regular OO model– there is no direct (implementaion level) composition operator to compose the *sync()* method to the various other methods. Instead, each of these methods explicitly call the *sync()* method, to achieve the desired behavior. As we discussed before, a naïve implementation may omit the additional references from the subclass methods that are now required.

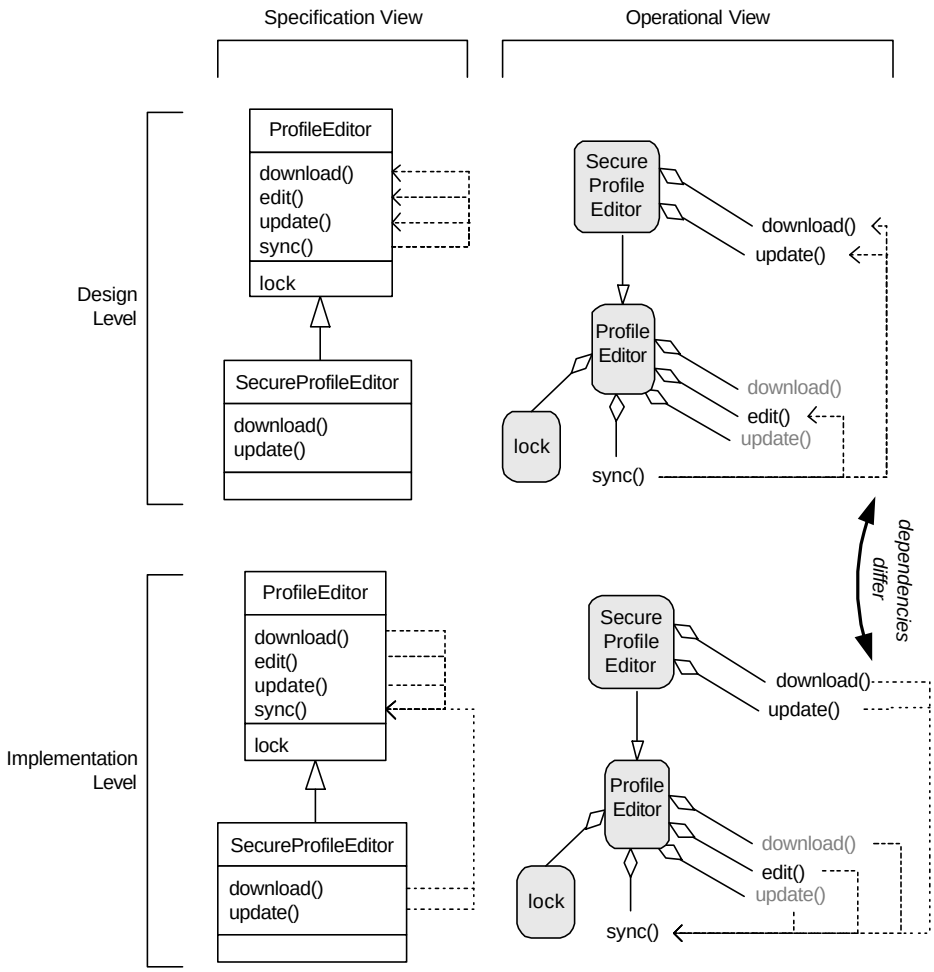


Figure 4. An overview of the various incarnations of a composition

From the specification view, it may not be clear that the newly created implementation has a problem. However, it turns out that going again to the operational view makes it straightforward to observe the problem(s). If we ignore the dotted dependencies (i.e. the ones that are added explicitly to *SecureProfileEditor*), we see that the dependencies of the implementation level and the design level operational views are really different. In principal, this implies that the behavior of the system will also be different.

When taking the dotted dependencies into account, the behavior of the system will be correct, but the directions of the dependencies have a direct impact on maintainability of the system! The specification/implementation of a dependency is always located at the origin of the dependency arrow. This means that the dependencies in the design level operational perspective can be mapped to a single one-to-many dependency located at the *sync()* method, whereas the dependencies in the implementation level operational perspective can only be expressed as five distinct one-on-one dependencies, distributed over the design, and hence much more sensitive to changes.

Concluding, by modeling a (sub-)system in the various perspectives of our conceptual model, focusing on the composition relations and the dependencies between elements of our design, we can precisely observe where and why the mapping from design level composition operators to implementation level composition operators can go wrong: not only from a functional perspective, but also from a maintainability point of view.

6. Related Work

As we discussed in section 5, the synchronization inheritance anomaly by Matsuoka et.al. [20][21] is the first explicit analysis of composition anomaly (but restricted to the domain of synchronization and composition through inheritance). In that case the functional behavior of objects and synchronization are the (only) two concerns involved, composition operators are synchronization specification mechanisms and inheritance. The analysis results in a categorization of typical examples where inheritance anomaly may appear. In our opinion, however, this does not reveal the source of the problem. In [13], the model and categorization of Matsuoka et.al. is adopted, generalized, and subsequently formalized. This hence offers a powerful tool to examine these categories of synchronization inheritance anomalies.

In [6], the synchronization inheritance anomaly was analyzed by investigating its origins, based on a canonical model of (object-oriented) synchronization mechanisms. This model served to explain all known occurrences of synchronization inheritance anomaly as limited separation of concerns and/or too strong coupling of elements in the canonical synchronization model. In [2], we showed that the notion of inheritance anomaly also applies to the domain of real-time constraint specifications. We introduced a canonical model of object-oriented real-time constraint specification mechanism, similar to the canonical synchronization model. These activities have provided a strong inspiration for the generalized notion of composition anomaly as introduced in this paper. In [4] we have shown that the composition filters model performs far better than the traditional object model in avoiding composition anomalies. In the composition filters model, aspects are represented by so-called composition filters, which modularly and orthogonally enhance the behavior of objects. We have developed models for various concerns, such as real-time, multiple views, coordinated behavior and error handling [1][2][3][6][8].

Various researchers carry out research on component oriented software development to address open systems requirements [23]. A component is defined as a static software abstraction that can be composed with other components to make an application. We think that our work can be used to reason about and control the composition of components during software design.

One of the important concerns in software engineering is the reuse of existing software components. Due to differences in signature, protocol or the semantics of the components, the reuse and as such the composition of these components can be very difficult. Research on interoperability aims to come up with various methods and techniques to improve interoperability and likewise to provide suitable composition schemes for the components [30].[17]

Aspect-oriented software development (AOSD) has received a lot of attention, and inspired a lot of research activity in the recent years. Its essential technological innovation is the introduction of a new kind of composition operator, commonly referred to as *aspect weaving*, that can express a certain modularization of concerns that was not possible with the well-known composition techniques before. We consider the growing interest in AOSD as a clear indication that composition techniques have significant practical relevance, and that improved composition techniques can avoid many undesirable (because hard to maintain) implementation structures.

The work in this paper has a clear relation with Model-Driven Architecture [16]: briefly, the design level concern model maps to Platform Independent Models (PIM), and the implementation level concern model is a Platform-Specific Model (PSM). The mapping of concepts from the design level to the implementation level is an example of a PIM-to-PSM transformation. The work in this paper is relevant in the MDA-context when reasoning about

the compositionality of models before and after transformation. One of the lessons learned in this paper that can be carried to the MDA domain is that the (composition) characteristics of the PSM can have an impact on the ability to further compose the results of a transformation.

One of the goals of this paper is to influence the designers of new composition schemes in languages and technology that are composition-intensive; to make them aware of the issues they need to take into consideration when introducing new composition operators. In particular, they ought to be aware that certain deficiencies and/or restrictions can be the cause of composition anomalies. Hence, as a follow-up to the work presented in this paper, we have explored the design space of composition operators and have proposed rules that composition operators need to fulfill in order to avoid the occurrence of composition anomaly.

7. Conclusion

This paper has the following contributions:

(a) It introduces a general framework for reasoning about software composition and defines the related terminology.

(b) It presents a set of concrete heuristics for reasoning about design-level composability.

(c) It defines the notion of composition anomaly, and shows how composition anomalies can affect the ability to compose systems.

(d) It explains the exact reasons when and why composition anomaly occurs.

In this paper, we have presented the groundwork for further analysis and better understanding of the issues in developing and composing modular software from multiple concerns.

To create components that are composable, components must be designed with composition in mind. This means that components must meet the five rules defined in section 4.

Composable components, however, do not guarantee a successful composition of multiple components. The composition scheme or the composition technique is a crucial factor in the realization of the composition. Depending on the characteristics of the composition scheme, a particular composition may suffer from *composition anomaly*, as defined in section 0.

ACKNOWLEDGEMENTS

We would like to thank Maurice Glandrup for his contributions during earlier stages of this work. This work has been partly funded by the European Union, in the scope of the EASYCOMP project, and by the Dutch Organization for Sciences (NWO), in the scope of the JACQUARD project Aspect-Oriented Software Architecture Design.

REFERENCES

[1] M. Aksit, L. Bergmans & S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, Proc. of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans & A. Yonezawa. Abstracting Object-Interactions Using Composition-Filters, In Object-based Distributed Processing, R. Guerraoui, O. Nierstrasz & M. Riveill (eds.), LNCS 791, Springer-Verlag, 1993, pp 152-184

[3] M. Aksit, J. Bosch, W. v.d. Sterren & L. Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters, Proc. of ECOOP '94, LNCS 821, Springer Verlag, July 1994, pp. 386-407

[4] M. Aksit & B. Tekinerdogan. Component composability issues in object-oriented programming, in Proc. of international Smalltalk Java conference, STJA '97, Erfurt, Germany, 1997.

[5] AspectJ Language Specification, XEROX Corporation, URL: http://www.aspectj.org, 1999

[6] L. Bergmans. Composing Concurrent Objects, Ph.D. thesis, University of Twente, The Netherlands, 1994

[7] L. Bergmans, Aspects of AOP: Scalability and application to domain modelling, position paper for the first 'Friends of AOP' workshop, XEROX PARC, Palo Alto, 1996

[8] L. Bergmans & M. Aksit, Composing Synchronization and Real-Time Constraints, Journal of Parallel and Distributed Programming, September 1996

[9] L. Bergmans, An Introduction to Composability, in the workshop report of the ECOOP'96 Workshop on Composability in Object-Oriented Programming, in [Mühlhaüser 97], 1997

[10] L. Bergmans & M. Aksit, Analyzing Multi-Dimensional Programming in AOP and Composition Filters, position paper for the OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns, 1999

[11] L. Bergmans & M. Aksit, Aspects & Crosscutting in Layered Middleware Systems, position paper for the workshop on Reflective Middleware, in conjunction with Middleware 2000, 2000

[12] M. Chi, J. Slotta, N. de Leeuw, From things to processes: A theory of conceptual change for learning science concepts. In Learning and Instructions, Vol. 4., pp. 27-43, Elsevier Science

[13] L. Crnogorac, A.S. Rao, K. Ramamohanarao, *Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming*, E. Jul (Ed.), Proceedings of ECOOP98, LNCS 1445, pp. 571-600, Springer Verlag, 1998

[14] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming. In proceedings of ECOOP '97, Springer-Verlag LNCS 1241. June 1997.

[16] A. Kleppe, J. Warmer, W. Bast. MDA Explained, The Model-Driven Architecture: Practice and Promise, Addision-Wesley, 2003.

[17] D. Konstantas. Interoperation of object-oriented applications. In O. Nierstrasz and D. Tsichritzis, eds., Object-Oriented Software Composition. Prentice-Hall, 1995.

[18] P. Koopman, A Taxonomy of decomposition Strategies based on Structures, Behaviors, and Goals, In Design Theory & Methodology '95, 1995

[19] Lane, T.G. Guidance for User Interface Architectures. in: M. Shaw, D.Garlan: Software Architecture, Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[20] S. Matsuoka, K. Wakita & A. Yonezawa, Synchronization Constraints with Inheritance: What is Not Possible- So What is?, Tokyo University, Internal Report, 1990

[21] S. Matsuoka & A. Yonezawa, Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Research Directions in Concurrent Object-Oriented Programming, (eds.) G. Agha, P. Wegner & A. Yonezawa, MIT Press, April 1993, pp. 107-150

[22] N. Michelena & P. Papalambros, A Hypergraph Framework for Optimal Model-Based Decomposition of Design Problems, Computational Optimization and Applications, Vol. 8, No.2, September 1997, pp. 173-196

[23] Nierstrasz, O. & Tsichritzis, D. (eds.), Object-Oriented Software Composition. Prentice Hall, 1995.

[24] Ossher, H., & Tarr, P. Multi-Dimensional Separation of Concerns using Hyperspaces. IBM Research Report 21452, April, 1999.

[25] D. Parnas. On the Criteria for Decomposing Systems into Modules. Communications of the ACM 15, 12 (December 1972): 1053-1058.

[26] H.A. Simon, The Sciences of the Artificial, 3rd edition, The MIT Press, Cambridge (MA), 1996.

[27] P. Tarr, H. Ossher, W. Harrison & S.M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In proceedings of ICSE 21, May, 1999.

[28] P. Tarr, M. D'Hondt, L. Bergmans & C. Lopes (eds.), "Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems For, Advanced Separation Of Concerns", ECOOP2000 workshop proceedings, LNCS series, Springer-Verlag, 2000

[29] B. Tekinerdogan. Synthesis-Based Software Architecture Design, PhD Thesis, Dept. of Computer Science, University of Twente, The Netherlands, 2000.

[30] A. Vallecilo, J. Hernandez & J.M. Troya. Object interoperability. In Object-Oriented Technology: ECOOP '99 Workshop Reader, number 1743 in LNCS, pages 1-21, Springer Verlag.

[31] Webster's Dictionary, http://www.m-w.com/, 2003.

[32] W.E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 2, April 1989, pp. 249-282